Pierre-Alexis Barras 40022016 Concordia University Fall 2020

# SOEN 422 Lab 1C Discussion

***Note:** for this Lab, ***raw C/C++*** was interpreted as disallowing higher level functions from the AVR libraries such as *pinMode(), digitalRead(),* etc. Register names however, such as *PORTB*, that are defined within the AVR libraries were permitted.

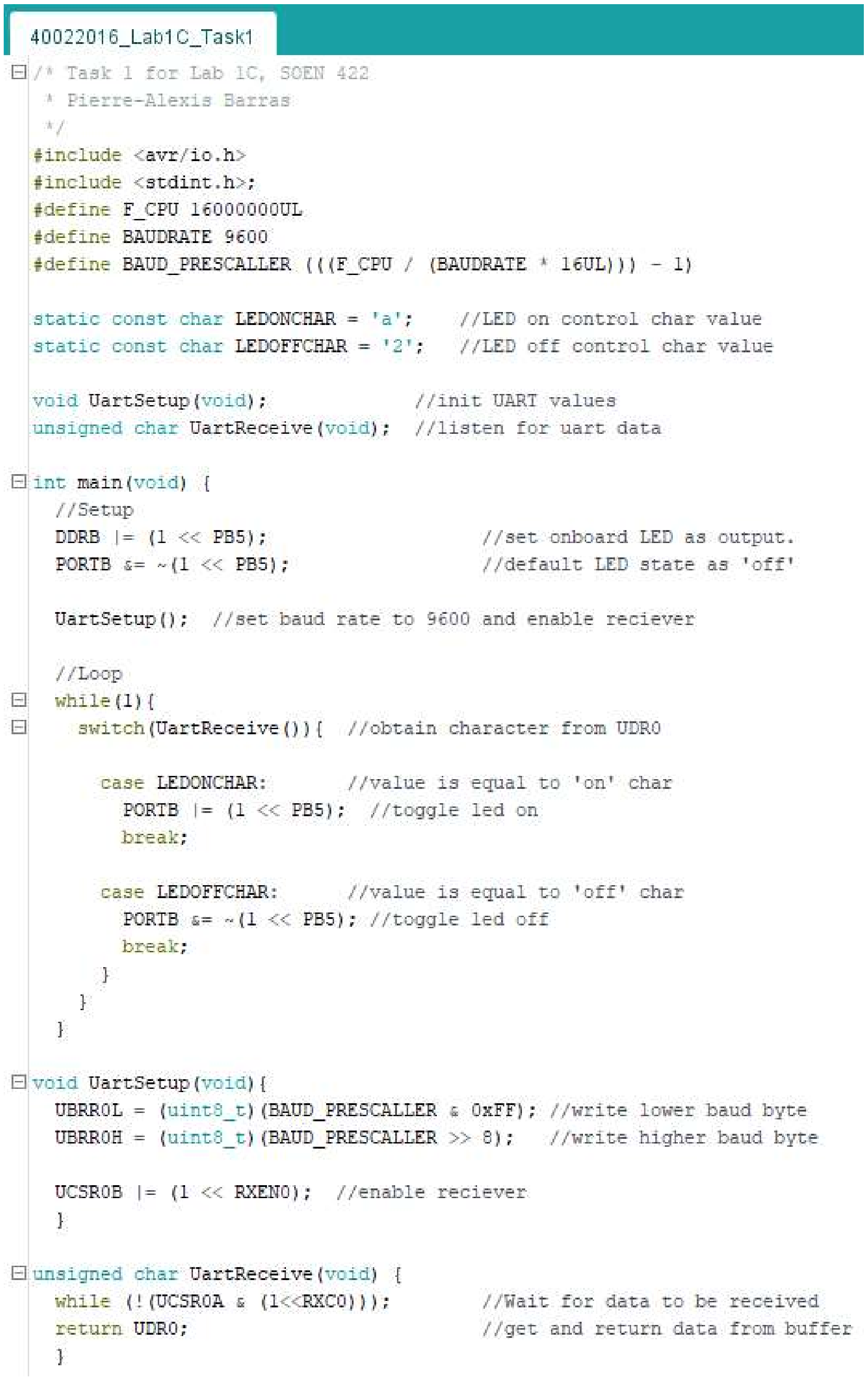
## Task 1

*Related file:* 40022016_Lab1C_Task1.ino

For the implementation of this task, the baud and surrounding values needed to be defined in order to make use of the ATmega328p's UART communication features. Taken from the ATmega328p datasheet were the values for the clock speed(16MHz), Baud rate (9600) and the equation for determining the prescalers. Next, two character values are instantiated as const values to turn on or off the led. As defined in the task, the 'on' character is set to 'a' and off to '2'. Defining the values this way allows users to customize the control characters by only modifying two values, which works towards a value of portability.

Next two functions are defined: UartSetup(), and UartReceive(). UartSetup begins by setting the baud rate registers (UBRR0L, UBRR0H) to the value defined by the BAUD_PRESCALLER, splitting appropriately the values across the high and low registers. Then the function enables the UART receiver by setting the RXEN0 on the UART control register UCSR0B. UartReceive is implemented by looping initially while waiting for there to be unread data within the buffer register. Then once the flag is set, it reads and returns the byte found in the buffer register as a character.

The main function of the implementation begins by setting the onboard led as an output and explicitly setting it as off by default. Then the main calls the UartSetup function to initialize UART communication. Next the loop of the function has a single switch case which check what is returned by UartReceive. If the 'on' character is received, then the led is toggled on. If the 'off' character is received, then the led is turned off. If any other character is detected, nothing occurs as no extra cases are defined. This was done to protect against whitespace characters, such as the newline character '\n', from affecting the functionality of the program.


40022016_Lab1C_Task1

```c
/* Task 1 for Lab 1C, SOEN 422
 * Pierre-Alexis Barras
 */
#include <avr/io.h>
#include <stdint.h>;
#define F_CPU 16000000UL
#define BAUDRATE 9600
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)

static const char LEDONCHAR = 'a';    //LED on control char value
static const char LEDOFFCHAR = '2';   //LED off control char value

void UartSetup(void);             //init UART values
unsigned char UartReceive(void);  //listen for uart data

int main(void) {
  //Setup
  DDRB |= (1 << PB5);                  //set onboard LED as output.
  PORTB &= ~(1 << PB5);                //default LED state as 'off'

  UartSetup();  //set baud rate to 9600 and enable reciever

  //Loop
  while(1){
    switch(UartReceive()){  //obtain character from UDR0

      case LEDONCHAR:       //value is equal to 'on' char
        PORTB |= (1 << PB5);  //toggle led on
        break;

      case LEDOFFCHAR:      //value is equal to 'off' char
        PORTB &= ~(1 << PB5); //toggle led off
        break;
      }
    }
  }

void UartSetup(void){
  UBRR0L = (uint8_t)(BAUD_PRESCALLER & 0xFF); //write lower baud byte
  UBRR0H = (uint8_t)(BAUD_PRESCALLER >> 8);   //write higher baud byte

  UCSR0B |= (1 << RXEN0);  //enable reciever
  }

unsigned char UartReceive(void) {
  while (!(UCSR0A & (1<<RXC0)));        //Wait for data to be received
  return UDR0;                          //get and return data from buffer
  }
```

## Task 2

*Related file:* 40022016_Lab1C_Task2.ino

Task 2 proved to be tricky to implement due to requirement to use dynamically allocated memory to print a greeting. The program uses three volatile byte-sized values to control the 'string' data: the character array "*dataString*", the integer "*slotCount*", and the integer "*index*". *"dataString"* is the character pointer array that will contain the memory locations of the characters to be stored and outputted to the user. *slotCount* describes the number of total available slots in memory allocated for dataString. *index* describes the current lowest unwritten position on dataString; it can be though of as the curser for writing to the character array. The reason these values were made volatile is because they must be modified within interrupt service routines.

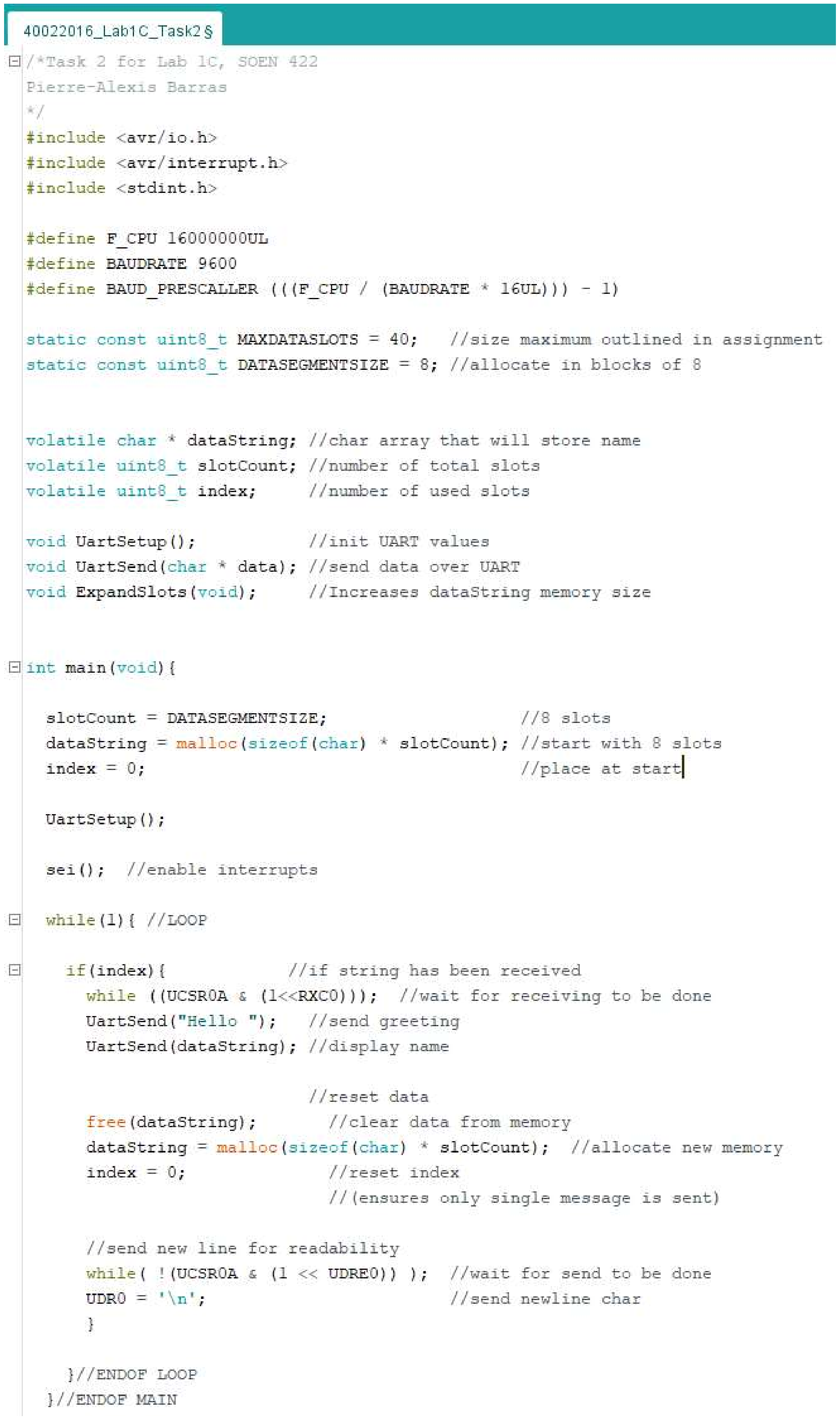
Next, the program also makes use of two static const integers to achieve this: *MAXDATASLOTS* set to 40 (defined in assignment), and *DATASEGMENTSIZE* set to 8 (factor of 40). The former is used to define how many character sized memory slots are added during a reallocation of the character array, and the latter providing the upper limit for the amount of memory slots permitted to be given to the character array.

The implementation uses three extra functions aside from the main function. The "*UartSetup*" function performs identically to the previous task's except it also enables the transmitter and reading interrupts. The "*UartSend*" function takes a character pointer array and iteratively sends the data until it hits an ASCII whitespace character, with the exception of the SPACE character (ASCII < 32). This is done to ensure that readable text data will be transmitted through UART. The "*expandSlots"* function is concerned with reallocating additional memory for dataString. Memory is added if the slotCount value is less than maximum defined by the MAXDATASLOTS constant. If memory is to be added, the slotCount variable is increased by an amount equal to the value defined by the DATASEGMENTSIZE constant, in this case 8. Then an amount of memory equal to the new value of slotCount is reallocated for dataString. If the maximum of permitted memory is reached, then the index is set to the MAXDATASLOTS value minus one, effectively placing it at the end of the dataString.

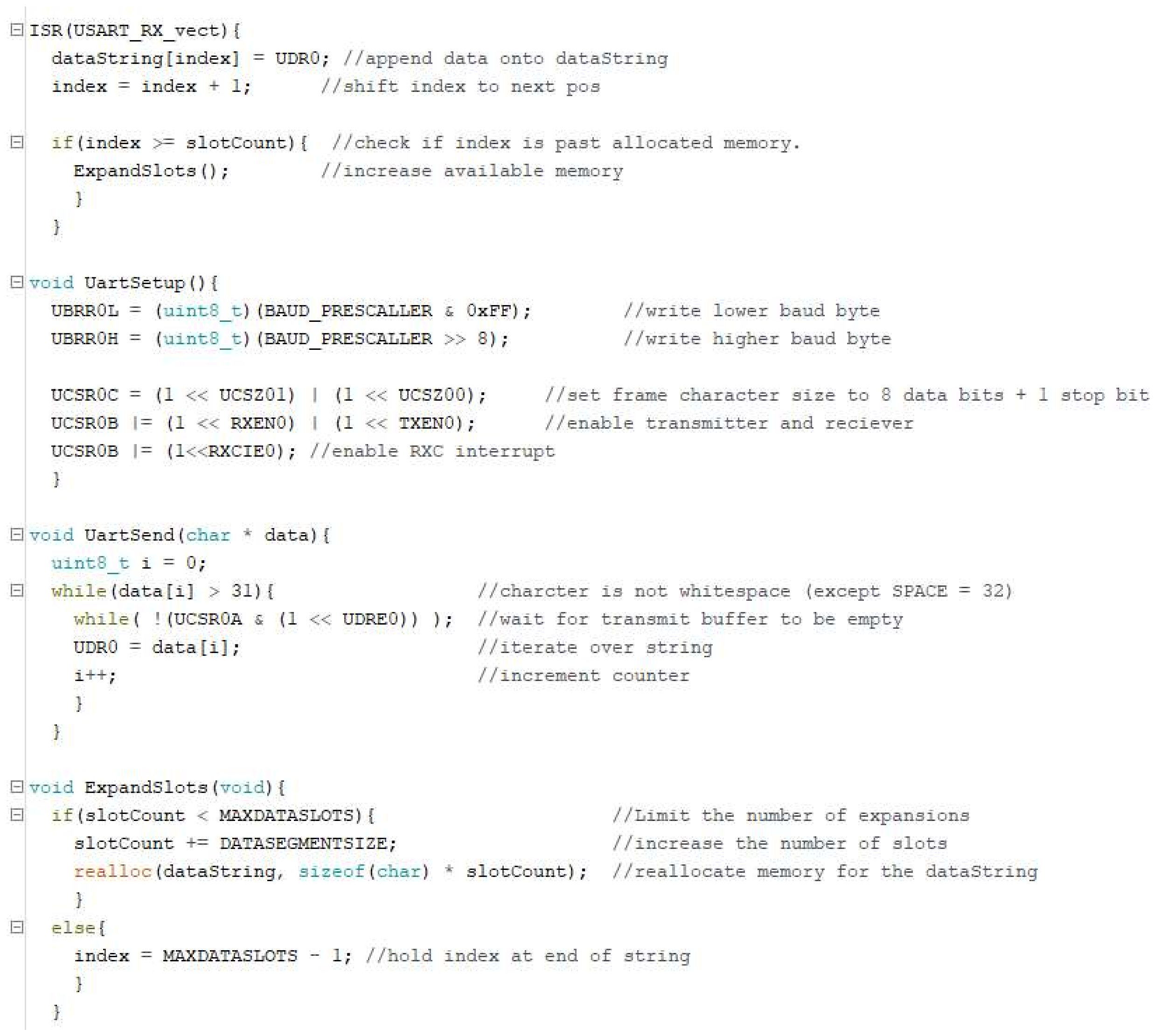
The interrupt service routine (ISR) implemented triggers when there is data to be read from the UART data register. The ISR begins by writing the char byte to dataString at the position defined by the index. Next, the index is incremented by one. If the index is now at the end of the available memory, by being greater than or equal to slotCount, then the expandSlots function is called, and memory is then added.

The main function begins by first allocating and initializing the volatile values. The slotCount integer is set equal to DATASEGMENTSIZE of 8, and the dataString is allocated that much character memory. The index is set to 0, to signify it being at the beginning of the dataString array. Next UartSetup is called and interrupts are enabled.
The loop of the program does nothing until the index is set to a non-zero value. In other words, if data was received over UART. If this occurs, the program waits for all data to finish being received and then sends two messages over UART through the UartSend function. First it sends a preliminary greeting string "Hello ", followed by the contents of dataString.

Afterwards the volatile data is reset. First the memory of dataString is freed through the “free” C function. Then dataString allocated fresh memory. This clears the string, preventing any residue data from being displayed, as well as stops any memory leaks. Finally, the index is set back to 0, placing it back to the beginning of the new dataString.


40022016_Lab1C_Task2 §

```
/*Task 2 for Lab 1C, SOEN 422
Pierre-Alexis Barras
*/
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>

#define F_CPU 16000000UL
#define BAUDRATE 9600
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)

static const uint8_t MAXDATASLOTS = 40;   //size maximum outlined in assignment
static const uint8_t DATASEGMENTSIZE = 8; //allocate in blocks of 8


volatile char * dataString; //char array that will store name
volatile uint8_t slotCount; //number of total slots
volatile uint8_t index;     //number of used slots

void UartSetup();           //init UART values
void UartSend(char * data); //send data over UART
void ExpandSlots(void);     //Increases dataString memory size


int main(void){

  slotCount = DATASEGMENTSIZE;                    //8 slots
  dataString = malloc(sizeof(char) * slotCount); //start with 8 slots
  index = 0;                                      //place at start

  UartSetup();

  sei();  //enable interrupts

  while(1){ //LOOP

    if(index){             //if string has been received
      while ((UCSR0A & (1<<RXC0)));  //wait for receiving to be done
      UartSend("Hello ");   //send greeting
      UartSend(dataString); //display name

                          //reset data
      free(dataString);       //clear data from memory
      dataString = malloc(sizeof(char) * slotCount);  //allocate new memory
      index = 0;              //reset index
                              //(ensures only single message is sent)

      //send new line for readability
      while( !(UCSR0A & (1 << UDRE0)) );  //wait for send to be done
      UDR0 = '\n';                        //send newline char
      }

    }//ENDOF LOOP
  }//ENDOF MAIN
```



```c
ISR(USART_RX_vect){
  dataString[index] = UDR0; //append data onto dataString
  index = index + 1;      //shift index to next pos

  if(index >= slotCount){  //check if index is past allocated memory.
    ExpandSlots();        //increase available memory
    }
  }

void UartSetup(){
  UBRR0L = (uint8_t)(BAUD_PRESCALLER & 0xFF);        //write lower baud byte
  UBRR0H = (uint8_t)(BAUD_PRESCALLER >> 8);          //write higher baud byte

  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);     //set frame character size to 8 data bits + 1 stop bit
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0);      //enable transmitter and reciever
  UCSR0B |= (1<<RXCIE0); //enable RXC interrupt
  }

void UartSend(char * data){
  uint8_t i = 0;
  while(data[i] > 31){                   //charcter is not whitespace (except SPACE = 32)
    while( !(UCSR0A & (1 << UDRE0)) );  //wait for transmit buffer to be empty
    UDR0 = data[i];                     //iterate over string
    i++;                                //increment counter
    }
  }

void ExpandSlots(void){
  if(slotCount < MAXDATASLOTS){                        //Limit the number of expansions
    slotCount += DATASEGMENTSIZE;                      //increase the number of slots
    realloc(dataString, sizeof(char) * slotCount);  //reallocate memory for the dataString
    }
  else{
    index = MAXDATASLOTS - 1; //hold index at end of string
    }
  }
```

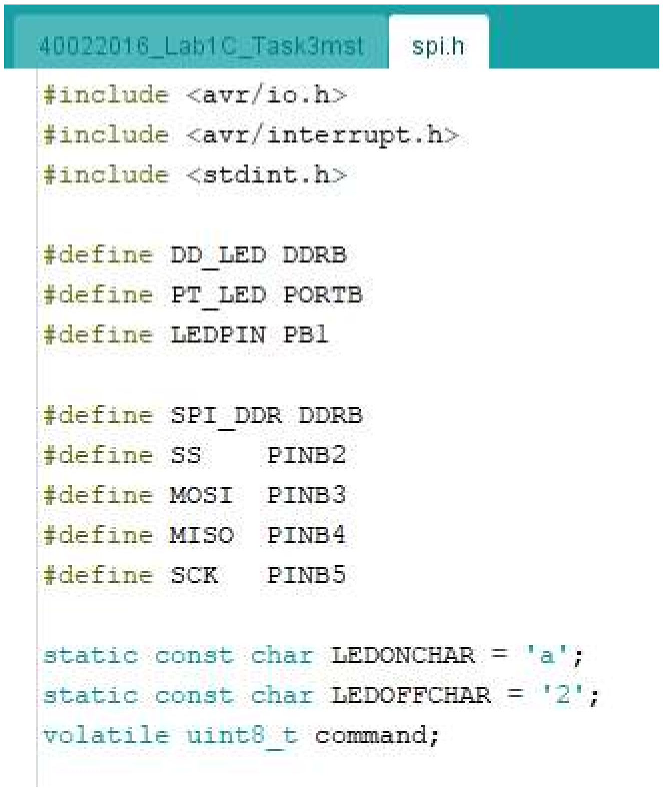
## Task 3

*Related files:* 40022016_Lab1C_Task3mst.ino
40022016_Lab1C_Task3slv.ino
spi.h

***Note:** Due to the nature of SPI communication and how the ATmega328p is integrated onto the Arduino Nano, the implementation cannot make use of the onboard LED while SPI communication is used as PB5 is occupied by the SPI clock. Thus, it necessitated the use of an external LED, which for this implementation was defined to be connected to *PB1* on the ATmega238p or *D9* on the Arduino Nano.

This implementation made use of three files: a common header file, a master device sketch, and a slave device sketch.

The header file “spi.h” contains the definitions to be used by both the Master device and Slave device. Notably, the registers and values related to the LED to be controlled and the data direction register and bit positions for SPI communication (MISO, MOSI, clock, and SS). This information is defined in the in a separate header file because it permits the code to be portable by only needing to modify this file.

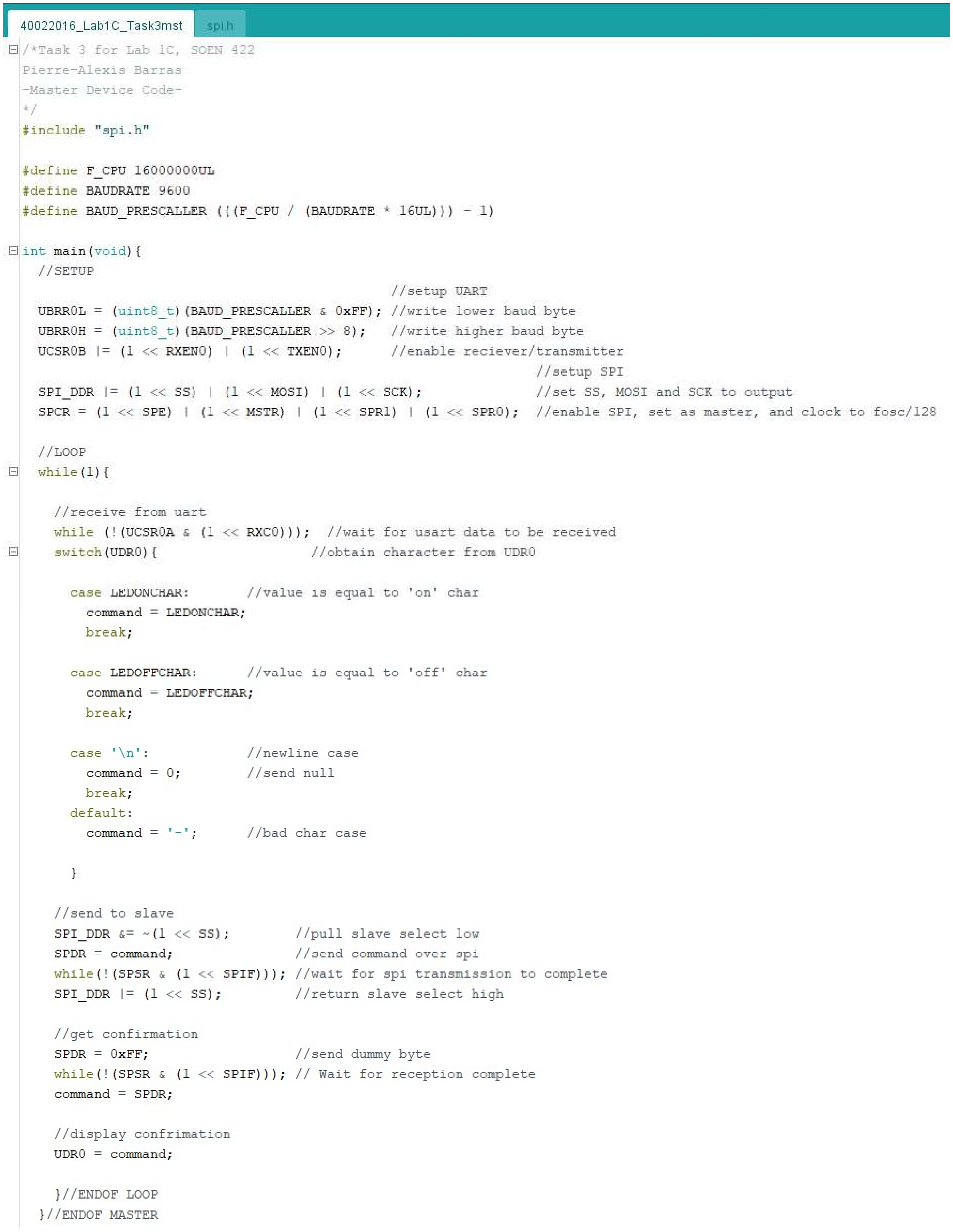


```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>

#define DD_LED DDRB
#define PT_LED PORTB
#define LEDPIN PB1

#define SPI_DDR DDRB
#define SS    PINB2
#define MOSI  PINB3
#define MISO  PINB4
#define SCK   PINB5

static const char LEDONCHAR = 'a';
static const char LEDOFFCHAR = '2';
volatile uint8_t command;
```

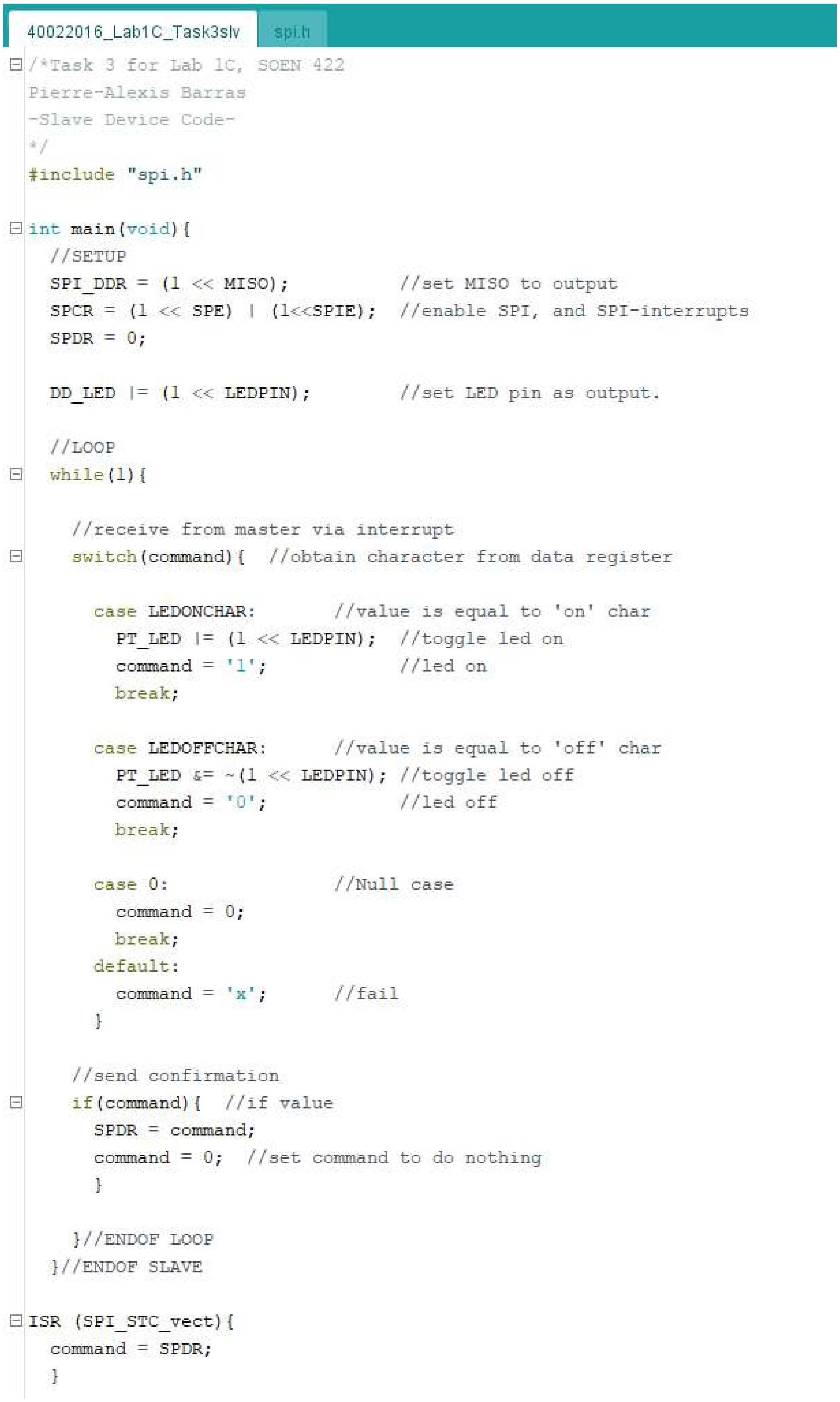
The master device code begins by setting up UART communication, followed by setting up the SPI registers for SPI master communications. This is achieved by setting the data direction register outlined in spi.h so that the slave select, MOSI, and clock pins are set to outputs. By default, the MISO pin is an input as is defined for all I/O pins on the Arduino Nano and ATmega328p. Next the SPI Control Register enables SPI communication, sets the device as master and sets the clock prescaler to 128.

The loop for the master device begins by waiting for an instruction character over UART. Once a character is received, it the appropriate command is sent to the slave. This starts by pulling the Slave Select low, then sending the command by loading the value into the SPDR register. Then the program waits for the transmission is complete before resetting the slave select bit. Next, the program sends a dummy byte to the slave as it waits for its response. Once the data is received, the program then transmits the data over UART to the user, displaying the success or failure of the inputted command (‘1’ -> LED was turned on, ‘0’ -> LED was turned off, ‘x’-> bad command was submitted and did nothing).

The slave device code begins by setting the SPI registers for SPI communication as a slave. It firstly sets the MISO pin as an output, followed by enabling SPI communications and SPI interrupts vectors. The SPI data register is then set to 0 to ensure that the register is empty. Finally, the LED PIN is set as an output.
The slave makes use of the interrupt service routine flagged by the SPI reception vector. When the ISR is triggered, it loads the contents of the SPI data register into the command data byte to be interpreted. The loop of the slave device code consists of two steps: interpreting the contents of the command byte and sending a confirmation. The interpretation consists of a switch-statement evaluating the current value of the command byte. If the command byte is equal to the LEDONCHAR, then the LED is turned on and the command value is set to the character value of ‘1’. Similarly, if the value is equal to the LEDOFFCHAR, the LED is turned off and the command value is set to the character value of ‘0’. In the case that null is evaluated, then the command byte is set to 0 and nothing is done. For any other character, nothing is done to the LED and the command byte is set to the character value of ‘x’, signifying a bad value is sent. Next if the command byte has a non-zero value, then it is sent to the master as a confirmation of the command being received. Then the command is set to 0 to prevent and repetitive sendings.



```
/*Task 3 for Lab 1C, SOEN 422
Pierre-Alexis Barras
-Master Device Code-
*/
#include "spi.h"

#define F_CPU 16000000UL
#define BAUDRATE 9600
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)

int main(void){
  //SETUP
                                             //setup UART
  UBRR0L = (uint8_t)(BAUD_PRESCALLER & 0xFF); //write lower baud byte
  UBRR0H = (uint8_t)(BAUD_PRESCALLER >> 8);   //write higher baud byte
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0);      //enable reciever/transmitter
                                                               //setup SPI
  SPI_DDR |= (1 << SS) | (1 << MOSI) | (1 << SCK);             //set SS, MOSI and SCK to output
  SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR1) | (1 << SPR0); //enable SPI, set as master, and clock to fosc/128

  //LOOP
  while(1){

    //receive from uart
    while (!(UCSR0A & (1 << RXC0)));  //wait for usart data to be received
    switch(UDR0){                   //obtain character from UDR0

      case LEDONCHAR:       //value is equal to 'on' char
        command = LEDONCHAR;
        break;

      case LEDOFFCHAR:      //value is equal to 'off' char
        command = LEDOFFCHAR;
        break;

      case '\n':            //newline case
        command = 0;        //send null
        break;
      default:
        command = '-';      //bad char case

      }

    //send to slave
    SPI_DDR &= ~(1 << SS);        //pull slave select low
    SPDR = command;               //send command over spi
    while(!(SPSR & (1 << SPIF))); //wait for spi transmission to complete
    SPI_DDR |= (1 << SS);         //return slave select high

    //get confirmation
    SPDR = 0xFF;                  //send dummy byte
    while(!(SPSR & (1 << SPIF))); // Wait for reception complete
    command = SPDR;

    //display confrimation
    UDR0 = command;

    }//ENDOF LOOP
  }//ENDOF MASTER
```



40022016_Lab1C_Task3slv spi.h

```
/*Task 3 for Lab 1C, SOEN 422
Pierre-Alexis Barras
-Slave Device Code-
*/
#include "spi.h"

int main(void){
  //SETUP
  SPI_DDR = (1 << MISO);          //set MISO to output
  SPCR = (1 << SPE) | (1<<SPIE);  //enable SPI, and SPI-interrupts
  SPDR = 0;

  DD_LED |= (1 << LEDPIN);        //set LED pin as output.

  //LOOP
  while(1){

    //receive from master via interrupt
    switch(command){  //obtain character from data register

      case LEDONCHAR:       //value is equal to 'on' char
        PT_LED |= (1 << LEDPIN);  //toggle led on
        command = '1';            //led on
        break;

      case LEDOFFCHAR:      //value is equal to 'off' char
        PT_LED &= ~(1 << LEDPIN); //toggle led off
        command = '0';            //led off
        break;

      case 0:               //Null case
        command = 0;
        break;
      default:
        command = 'x';      //fail
      }

    //send confirmation
    if(command){  //if value
      SPDR = command;
      command = 0;  //set command to do nothing
      }

    }//ENDOF LOOP
  }//ENDOF SLAVE

ISR (SPI_STC_vect){
  command = SPDR;
  }
```